Linear cryptoanalysis lab2

8 червня 2024 р.

[1]:
```
import subprocess
from itertools import product
from math import prod
from random import randrange
import random
import numpy as np
import pandas as pd
from multiprocess import Pool
```

1 Комп’ютерний практикум №1

Виконано студентами

групи ФІ-32мн

Карловський Володимир

Кріпака Ілля

1.1 Мета лабораторної роботи

Практично ознайомитися із сучасним методом криптоаналізу блокових шифрів, набути навички у дослідженні стійкості блокових шифрів до лінійного криптоаналізу.

1.2 Постановка задачі

Постановка задачі	Зроблено
Реалізувати функції шифру Хейса	+
Пошук високоімовірних п’ятираундових лінійних апроксимацій шифру Хейса	+
Реалізувати атаку на перший раундовий ключ Хейса	+

1.3 Хід роботи / Опис труднощів

Позаяк другий практикум є логічним продовженням першого, тож реалізація була легшою, але із іншого не менш простою чим у першій лабораторній роботі. Проблеми виникли із:

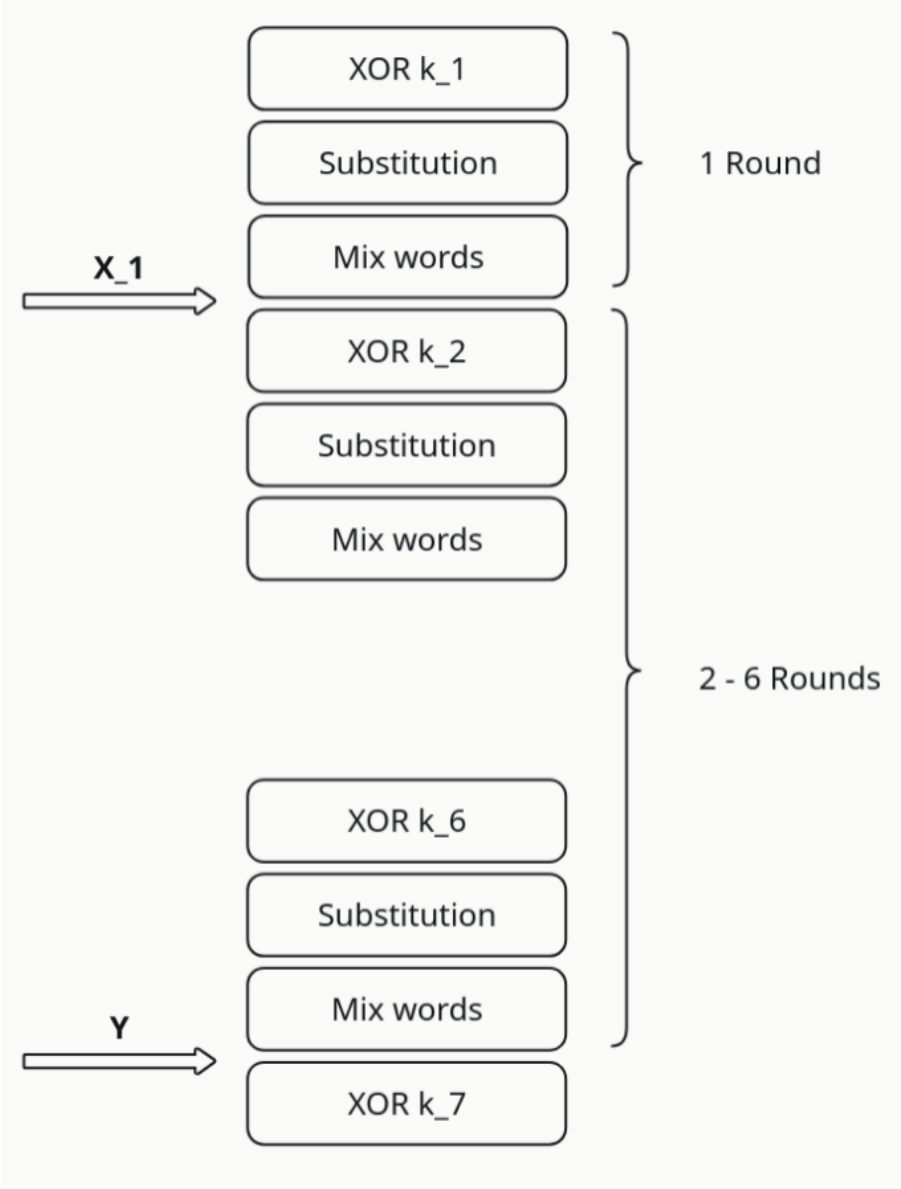
- написанням функції для перебору потенційних кадидатів у ключі. Як виявилося нічого складного, просто саме у той час наклалися часові проблеми;

- обчислювальними можливостями ноутбука. На пристрої де була запущена атака, використовувалися майже усі наявні можливості із оперативно пам’яттю. Майже завжди під час перебору оперативна пам’ять була заповнена на 50 гігабайт із 64. Звичайно можна було оптимізувати перебір та не використовувати звичайні *list()*, а навпаки *nd_array*, але то може у майбутньому зробимо таке покращення.

1.4 Варіант 5

S = (F, 8, E, 9, 7, 2, 0, D, C, 6, 1, 5, B, 4, 3, A)

1.5 Пояснення алгоритму



Алгоритм М2.

0. Нехай для r-раундового шифру E відома лінійна апроксимація його останніх $r-1$ раундів $\alpha \cdot X_1 \oplus \beta \cdot X_r = 0$ із суттєвою кореляцією ε. Алгоритм дозволяє знайти істинне значення ключа першого раунду шифрування k_1.

1. Одержати N пар відкритих та шифрованих текстів (X, Y), де $Y = E_K(X)$.

2. Для кожного кандидата k у ключі k_1 виконати такі дії:

2.1. Зашифрувати відкриті тексти X на один раунд: $X_1 = F_1(X, k)$.

2.2. Обчислити значення

$$\hat{u}(k) = \left|\{(X_1, Y) : \alpha \cdot X_1 \oplus \beta \cdot Y = 0\}\right| - \left|\{(X_1, Y) : \alpha \cdot X_1 \oplus \beta \cdot Y = 1\}\right|.$$

3. Ключ k_1 визначається як $k_1 = \arg\max_k |\hat{u}(k)|$.

Сам алгоритм можна розбити на такі частини.

1) Передобчислення таблиці коефіцієнтів кореляції лінійної апроксимації булевої функції для кожного значення α, β.
2) Знаходження високоймовірних п'ятираундових апроксимацій шифру Хейса із великим потенціалом.
3) Знаходження першого раундового ключа:
   (а) Створення даних для аналізу потенційних ключів. Тобто на цьому кроці генерується випадковий текст, який буде зашифровано двома різними шляхами. Перший із них буде зашифровано лише на один раунд, а другий зашифровано за допомогою виконуваного файлу "heys.bin".
   (б) Перебір усіх можливих ключів. На цьому кроці використовується атака М2, що проілюстрована на рисунку 2. Загалом атаку М2 можна описати наступним чином:
      i. Згенерувати один великий текст для перевірки ключів та зашифрувати його двома способами (один - зашифрувати на 1 раунд, другий - зашифрувати шифром Хейса).
      ii. Обчислюємо значення лінійної апроксимації і відповідне значення $\hat{u_k}$. Саме цю формулу можна трохи оптимізувати.

$$\begin{aligned}\hat{u_k} &= |(X_1, Y) : \{\alpha \cdot X_1 \oplus \beta \cdot Y = 0\}| - |(X_1, Y) : \{\alpha \cdot X_1 \oplus \beta \cdot Y = 1\}| \\ &= \#\text{ символів у}(\alpha \cdot X_1 \oplus \beta \cdot Y) - 2 \cdot |(X_1, Y) : \{\alpha \cdot X_1 \oplus \beta \cdot Y = 1\}|\end{aligned}$$

      iii. Збираємо певну кількість ключів, що мають максимальне значення $\hat{u_k}$.
      iv. У результаті залишиться деяка множина гіпотетичних ключів, перебираючи які можна буде дізнатися правильний ключ.

1.6 Порогові значення

- Порогове значення для методу гілок та границь: 0.00015. Чому саме таке значення? Експериментальним шляхом визначили, що саме після 0.00015 починає генеруватися дуже багато значень із маленькими ймовірностями.
- Кількість текстів потрібна для атаки: 1 шт у якому було 7000 екземплярів.
- Кількість ключів, що обираються на кожному кроці під час перевірки ключів для подальшого аналізу: 200. Це значення було обрано із розрахунку наявних обчислювальних можливостей, взагалі можна було 50 чи 100 взяти.

[2]:
```
s_block = [0xF, 0x8, 0xE, 0x9, 0x7, 0x2, 0x0, 0xD, 0xC, 0x6, 0x1, 0x5,
  ↪0xB, 0x4, 0x3, 0xA]
s_block_inverse = [0x6, 0xA, 0x5, 0xE, 0xD, 0xB, 0x9, 0x4, 0x1, 0x3,
  ↪0xF, 0xC, 0x8, 0x7, 0x2, 0x0]
ENC = True
DEC = False
MAX_16BIT_NUM = (1 << 16) - 1
VARIANT = 5


class heys:

    def heys_round(self, n, key=0):
        ct = n ^ key
        ct = self.substitute(ct, s_block)
        ct = self.mix_words(ct)
        return ct

    # r = (s_1(y_1), s_2(y_2), ... , s_n(y_n))
    def substitute(self, n, enc):
        s = s_block if enc == True else s_block_inverse
        r = 0
        for i in range(4):
            r |= s[(n >> (i * 4)) & 15] << (i * 4)
        return r

    # i-тий біт j-того фрагменту стає j-тим бітом i-того фрагменту.
    def mix_words(self, n):
        r = 0
        for j in range(4):
            for i in range(4):
                r |= (n >> (4 * j + i) & 1) << (4 * i + j)
        return r
```

1.7 Створення таблиці лінійних потенціалів

[3]:
```
heys_obj = heys()
# таблиця лінійних потенціалів для перестановки шифру Хейса
linear_potential = np.zeros((16, 16))


# makes AND operation to get what bits has to be XORed and performs
  ↪XOR of all
#  variables by taking weight of number and mod 2
def custom_mul(x: int, y: int):
    return (x & y).bit_count() % 2
```

```
for alpha in range(16):
    for beta in range(16):
        for x in range(16):
            linear_potential[alpha][beta] += (-1) ** (
                    custom_mul(alpha, x) ^ custom_mul(beta, heys_obj.
  ↪substitute(x, ENC)))
linear_potential /= 16
linear_potential **= 2

pd.DataFrame(linear_potential)
```

[3]:

	0	1	2	3	4	5	6	7	8	\
0	1.0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	
1	0.0	0.0625	0.0625	0.2500	0.0000	0.0625	0.0625	0.0000	0.0000	
2	0.0	0.0625	0.0625	0.2500	0.0625	0.0000	0.0000	0.0625	0.0000	
3	0.0	0.2500	0.0000	0.0000	0.0625	0.0625	0.0625	0.0625	0.2500	
4	0.0	0.0000	0.0625	0.0625	0.0625	0.0625	0.2500	0.0000	0.0625	
5	0.0	0.0625	0.0000	0.0625	0.0625	0.0000	0.0625	0.2500	0.0625	
6	0.0	0.0625	0.0000	0.0625	0.0000	0.0625	0.0000	0.0625	0.0625	
7	0.0	0.0000	0.0625	0.0625	0.0000	0.0000	0.0625	0.0625	0.0625	
8	0.0	0.0000	0.0000	0.0000	0.0000	0.2500	0.2500	0.0000	0.0625	
9	0.0	0.0625	0.0625	0.0000	0.2500	0.0625	0.0625	0.0000	0.0625	
10	0.0	0.0625	0.0625	0.0000	0.0625	0.2500	0.0000	0.0625	0.0625	
11	0.0	0.2500	0.0000	0.0000	0.0625	0.0625	0.0625	0.0625	0.0625	
12	0.0	0.0000	0.0625	0.0625	0.0625	0.0625	0.0000	0.0000	0.2500	
13	0.0	0.0625	0.2500	0.0625	0.0625	0.0000	0.0625	0.2500	0.0000	
14	0.0	0.0625	0.2500	0.0625	0.0000	0.0625	0.0000	0.0625	0.0000	
15	0.0	0.0000	0.0625	0.0625	0.2500	0.0000	0.0625	0.0625	0.0000	

	9	10	11	12	13	14	15
0	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
1	0.0625	0.0625	0.0000	0.2500	0.0625	0.0625	0.0000
2	0.0625	0.0625	0.2500	0.0625	0.0000	0.0000	0.0625
3	0.0000	0.0000	0.0000	0.0625	0.0625	0.0625	0.0625
4	0.0625	0.0000	0.2500	0.0000	0.0000	0.0625	0.0625
5	0.0000	0.0625	0.0000	0.2500	0.0625	0.0000	0.0625
6	0.0000	0.0625	0.0000	0.0625	0.2500	0.0625	0.2500
7	0.0625	0.0000	0.0000	0.0625	0.0625	0.2500	0.2500
8	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625
9	0.0000	0.2500	0.0625	0.0625	0.0000	0.0000	0.0625
10	0.2500	0.0000	0.0625	0.0000	0.0625	0.0625	0.0000
11	0.0625	0.0625	0.0625	0.0000	0.2500	0.0000	0.0000
12	0.0000	0.0625	0.0625	0.0625	0.0625	0.2500	0.0000
13	0.0625	0.0000	0.0625	0.0625	0.0000	0.0625	0.0000
14	0.0625	0.2500	0.0625	0.0000	0.0625	0.0000	0.0625
15	0.2500	0.0625	0.0625	0.0000	0.0000	0.0625	0.0625

[11]:
```
PROBABILITY_THRESHOLD = 0.00015
local_diff_table = {}


# betas
def get_new_betas(alpha):
    # obtain raw alpha values from one big Alpha
    alpha_list = [(alpha >> (4 * i)) & 15 for i in range(4)]
    # obtain non-null betas from s block diff table
    non_null_betas = []
    for i, alpha in enumerate(alpha_list):
        # filter betas with non-zero entries
        betas = [beta for beta in range(16) if
 ↪linear_potential[alpha][beta] != 0]
        non_null_betas.append(betas)

    betas = {}
    # generate all possible combinations that can appear in resulted
 ↪beta
    for (i, beta_list) in enumerate(product(*non_null_betas,
 ↪repeat=1)):
        beta = sum([beta_list[i] << (i * 4) for i in range(4)])
        betas[beta] =
 ↪prod([linear_potential[alpha_list[i]][beta_list[i]] for i in
 ↪range(4)])

    return betas


def differential_search(alpha):
    # shared between workers variable
    global local_diff_table

    # Γ_{t-1}(alpha) (Γ spells like 'g')
    g_prev = {alpha: 1}
    # like in rust -- 1..=5
    for i in range(1, 6):
        g_current = {}
        # misuse of 'beta' naming (by now we are iterating over last
 ↪alphas right before calculation of probabilities)
        for beta, p_i in g_prev.items():

            if beta not in local_diff_table:
                local_diff_table[beta] = get_new_betas(beta)

            # extract possible candidates for specific beta ('alpha')
 ↪and iterate over them
```

```
            for gamma in local_diff_table[beta]:
                tmp_prob = p_i * local_diff_table[beta][gamma]
                if gamma not in g_current:
                    g_current[gamma] = tmp_prob
                else:
                    g_current[gamma] += tmp_prob

        # Γ_{t}(alpha)
        g_new_filtered = {}
        #check ”bounds” and write to new list mixed words (aka gammas
↪are becoming new alphas)
        for gamma in g_current.keys():
            if g_current[gamma] > PROBABILITY_THRESHOLD:
                g_new_filtered[heys_obj.mix_words(gamma)] =
↪g_current[gamma]

        g_prev = g_new_filtered

    return (alpha, g_prev)
```

1.8 Знайдені за допомогою методу «гілок та границь» високоймовірних ланійних апроксимацій

Вони наведені у формі $(\alpha \to [(\beta, prob) \dots], \dots)$

[12]:
```
%%time

def prepare_alphas():
    alfas = []
    for i in range(4):
        for j in range(1, 16):
            alfas.append(j << (4 * i))
    return alfas


def init_worker(shared_diff_table):
    global local_diff_table
    local_diff_table = shared_diff_table


with Pool(initializer=init_worker, initargs=(local_diff_table,)) as
↪pool:
    candidates = list(
        pool.map(
            differential_search,
            prepare_alphas()
        )
```

```
    )

print('Possible candidates after ”branches and bounds” search,
  ↪candidates len: {0}\n candidates: \n'.format(
    len(candidates)))
list(candidates)
```

```
Possible candidates after ”branches and bounds” search, candidates len:
  ↪60
 candidates:

CPU times: user 340 ms, sys: 56.2 ms, total: 396 ms
Wall time: 1min 12s
```

[12]:
```
[(1,
  {17: 0.00018454398377798498,
   257: 0.00023846482508815825,
   272: 0.00026313314447179437,
   4352: 0.00019629453890956938,
   4096: 0.00016126653645187616,
   136: 0.0002424493432044983,
   2056: 0.0003714263439178467,
   2176: 0.00040375441312789917,
   34816: 0.000280916690826416,
   34944: 0.00020729750394821167,
   32904: 0.00019574910402297974,
   32768: 0.00022634863853454 59}),
 (2,
  {17: 0.0001951456069946289,
   257: 0.00017428936553187668,
   272: 0.000217383523704 4841,
   4352: 0.00019179822993464768,
   136: 0.00024132616817951202,
   2056: 0.0002853255718946457,
   2176: 0.00031377002596855164,
   34816: 0.0002876501530408859,
   34824: 0.00016975030303001404,
   34944: 0.00018735788762569427,
   32768: 0.00019478239119052887}),
 (3,
  {1: 0.00017478116205893457,
   17: 0.0001899251656141132,
   257: 0.000169116334291175,
   4352: 0.0002463887503836304,
   4353: 0.00020746083464473486,
   4368: 0.00017448668950237334,
   273: 0.0001525094558019191,
```

```
  4369: 0.00018956256099045277,
  4096: 0.0002558662963565439,
  8: 0.0002433881163597107,
  136: 0.00017390772700309753,
  2184: 0.00023550353944301605,
  34816: 0.0002921987324953079,
  34952: 0.00028022192418575287,
  2056: 0.00019656307995319366,
  2176: 0.00015114806592464447,
  32768: 0.00039661675691604614,
  34824: 0.00030264444649219513,
  34944: 0.000248810276389122}),
(4, {2176: 0.00017223507165908813, 34816: 0.00018981285393238068}),
(5,
 {257: 0.00015512356185354292,
  2056: 0.00024670176208019257,
  2176: 0.00023101642727851868,
  34816: 0.00020129792392253876,
  34824: 0.00016969069838523865,
  34952: 0.0001562647521495819,
  32768: 0.00016790814697742462}),
(6, {34816: 0.00015433132648468018}),
(7, {2176: 0.00015532225370407104}),
(8, {}),
(9,
 {17: 0.00018136418657377362,
  257: 0.0001720856234896928,
  272: 0.0002116433170158416,
  4352: 0.00019820188754238188,
  4353: 0.0001779826416168362,
  4113: 0.00018209032714366913,
  4369: 0.00021524858311749995,
  4096: 0.0001523562823422253,
  8: 0.00024153105914592743,
  136: 0.00019145570695400238,
  2056: 0.0002172808162868023,
  2176: 0.00022408505901694298,
  32896: 0.00019532954320311546,
  34816: 0.0002581402659416199,
  34824: 0.0003184061497449875,
  34944: 0.00016088970005512238,
  2048: 0.00015342188999056816,
  2184: 0.0001823841594159603,
  32904: 0.00017475849017500877,
  34952: 0.0002881605178117752,
  32768: 0.00026363832876086235}),
(10,
```

```
 {17: 0.00015406523016281426,
  272: 0.00016616898938082159,
  4113: 0.00017923925770446658,
  4096: 0.00016818076255731285,
  136: 0.00018798932433128357,
  2056: 0.00016310252249240875,
  2176: 0.00019206292927265167,
  34816: 0.0001660902053117752,
  34824: 0.00016141310334205627,
  34944: 0.00019028782844543457,
  32904: 0.0002344418317079544,
  34952: 0.00016515143215656 28,
  32768: 0.00025040097534656525}),
(11,
 {1: 0.00015845528105273843,
  17: 0.00019977326155640185,
  4352: 0.00018359377281740308,
  4368: 0.00015289310249499977,
  4096: 0.000221235619392246,
  8: 0.00021870806813240 05,
  136: 0.00021578185260295868,
  34816: 0.0002246331423521042,
  34824: 0.00018320418894290924,
  34944: 0.00022694095969200134,
  32904: 0.00016492046415805817,
  34952: 0.00016640126705169678,
  32768: 0.00034087523818016 05}),
(12,
 {4352: 0.00019735650857910514,
  4353: 0.00015870921197347343,
  4369: 0.00016025893273763359,
  8: 0.00015087611973285675,
  136: 0.00020255334675312042,
  2056: 0.00017717480659484863,
  2176: 0.00017203576862812042,
  34816: 0.00030050426721572876,
  34824: 0.00025399215519428253,
  2184: 0.0002118125557899475,
  34952: 0.00025476701557636 26}),
(13,
 {1: 0.00020089519966859 37,
  17: 0.00018900632858276367,
  272: 0.00020112040874664672,
  4352: 0.00015520621673204005,
  4113: 0.0001824386308726389,
  4096: 0.00016519577548024245,
  8: 0.00028971722349524 5,
```

```
  136: 0.0001929197460412979,
  2056: 0.00020887923892587423,
  2176: 0.00024209951516240835,
  34816: 0.00019564665853977203,
  34824: 0.00020011095330119133,
  34944: 0.0001684725284576416,
  32904: 0.00018364412244409323,
  32768: 0.00029943918343633413}),
(14,
 {1: 0.00017024693079292774,
  17: 0.00019836239516735077,
  4352: 0.0001641431008465588,
  4113: 0.0001688990741968155,
  4096: 0.00018759933300316334,
  8: 0.0002943561412394047,
  136: 0.00020356476306915283,
  2056: 0.00017611973453313112,
  2176: 0.0002341336803510785,
  34816: 0.00024613365530967 71,
  34824: 0.00015527987852692604,
  34944: 0.0001573469489812851,
  2048: 0.00017101794946938753,
  32904: 0.0001898306654766202,
  32768: 0.00030671909917145 97}),
(15,
 {17: 0.0001818088931031525,
  272: 0.00022080540657043457,
  4113: 0.00022286176681518555,
  4369: 0.00017271866090595722,
  8: 0.0001781051978468895,
  136: 0.0002074018120765686,
  2056: 0.0002441375982016325,
  2176: 0.000287030590698123,
  32776: 0.00015050615184009075,
  34816: 0.00015210360288619995,
  34824: 0.0002616783604025841,
  32904: 0.00027063558809459 21,
  34952: 0.00024138391017913818,
  32768: 0.00017101387493312 36}),
(16,
 {17: 0.00021915417164564133,
  257: 0.00029497360810637474,
  272: 0.00031525129452347755,
  4352: 0.000245650764554739,
  4368: 0.00018599489703774452,
  4113: 0.00016224663704633713,
  4096: 0.00020799599587917328,
```

```
  2056: 0.00023992641945369542,
  2176: 0.00026104532298631966,
  34816: 0.00017905517597682774}),
(32,
 {17: 0.0002774473978206515 3,
  257: 0.0002377253258600831,
  272: 0.00027033803053200245,
  4352: 0.00031538738403469324,
  4353: 0.00015954615082591772,
  4368: 0.00016875937581062317,
  4096: 0.00018469546921551228,
  136: 0.00015256553888320923,
  2056: 0.00016825205239001662,
  2176: 0.00020383602532092482,
  34816: 0.00018059487047139555}),
(48,
 {1: 0.00020819343626499176,
  17: 0.00020156893879175186,
  257: 0.00018382351845502853,
  272: 0.00015096738934516907,
  4352: 0.00027464982122182846,
  4353: 0.00023820530623197556,
  4368: 0.00020131096243858337,
  273: 0.00017673149704933167,
  4369: 0.00021626800298690796,
  4096: 0.00030543468892574 31,
  8: 0.00015944834740366787,
  2184: 0.00015672484005335718,
  34816: 0.00019659993995446712,
  34952: 0.00018694548634812236,
  32768: 0.000259537817328237,
  34824: 0.00019870209507644176,
  34944: 0.00016448991664219648}),
(64, {272: 0.0001531316665932536, 4352: 0.00018257391639053822}),
(80,
 {257: 0.00021313305478543043,
  272: 0.0001992222387343645,
  4352: 0.00016889150720089674}),
(96, {}),
(112, {}),
(128, {}),
(144,
 {1: 0.00022231256298255175,
  17: 0.00020458456128835678,
  257: 0.00021873572768527083,
  272: 0.0002475964829500299,
  4112: 0.00019207242803531 7,
```

```
  4352: 0.0002496638335287571,
  4353: 0.0002791935548884794,
  256: 0.00015575271027046256,
  273: 0.0001722361885185819,
  4113: 0.00021117152573424391,
  4369: 0.0002595032565295696,
  4096: 0.00023622875960427336,
  34952: 0.00018768863810691983,
  34824: 0.00016153710021172}),
(160,
 {17: 0.00018123607151210308,
  257: 0.00019550928846001625,
  272: 0.00019833468832075596,
  4352: 0.00017225323244929314,
  4353: 0.00016397261060774326,
  4368: 0.00019596191123127937,
  4113: 0.00021781958639621735,
  4369: 0.00015810714103281498,
  4096: 0.00024809944443404675}),
(176,
 {1: 0.0001925964024849236,
  17: 0.0002206650678999722,
  4352: 0.00020752224372699857,
  4353: 0.0001526575069874525,
  4368: 0.0001809120294637978,
  4113: 0.00015878723934292793,
  4096: 0.0002709709224291146,
  32768: 0.00020480824330777}),
(192,
 {17: 0.0001798509620130062,
  257: 0.0001588389277458191,
  272: 0.0001560836099088192,
  4352: 0.00023890100419521332,
  4353: 0.00018653785809874535,
  273: 0.00015106238424777985,
  4369: 0.00018805405125021935,
  34816: 0.0001975955383386463,
  34824: 0.0001619462709641084,
  34952: 0.00016277235408779234}),
(208,
 {1: 0.00026186337345279753,
  17: 0.00021392712369561195,
  257: 0.0001846187878982164,
  272: 0.00023365245579043403,
  4352: 0.00019230111502110958,
  4353: 0.0001547880528960377,
  4113: 0.00020241147285560146,
```

```
  4096: 0.00025081091007450595,
  8: 0.000151603773701936,
  2176: 0.00016403298650402576}),
(224,
 {1: 0.000282736262306571,
  17: 0.0002296827733516693,
  272: 0.00018548741354607046,
  4352: 0.00025188352447003126,
  4353: 0.00015065993648022413,
  256: 0.00016949904966168106,
  4113: 0.00018912050290964544,
  4096: 0.00025994572206400335}),
(240,
 {1: 0.00017735055007506162,
  17: 0.00021739088697358966,
  257: 0.0002334243072255049,
  272: 0.00027415436125011183,
  4097: 0.00015068624998093583,
  4352: 0.00018406828166916966,
  4353: 0.00024815382494125515,
  4113: 0.0002650000424182508,
  4369: 0.00022795714903622866,
  4096: 0.00016408070587203838,
  136: 0.00015318437363021076,
  2176: 0.00018366333097219467,
  32904: 0.00017938390374183655,
  34952: 0.000155067886225879 2}),
(256, {}),
(512, {}),
(768, {}),
(1024, {}),
(1280, {}),
(1536, {}),
(1792, {}),
(2048, {}),
(2304, {}),
(2560, {}),
(2816, {}),
(3072, {}),
(3328, {}),
(3584, {}),
(3840, {}),
(4096, {}),
(8192, {}),
(12288, {}),
(16384, {}),
(20480, {}),
```

```
 (24576, {}),
 (28672, {}),
 (32768, {}),
 (36864, {}),
 (40960, {}),
 (45056, {}),
 (49152, {}),
 (53248, {}),
 (57344, {}),
 (61440, {})]
```

1.9 Відфільтровані кандидати

Вони наведені у формі $((\alpha, \beta), prob)$

[13]:
```
filtered_candidates = []
for alpha, betas in list(candidates):
    for beta, prob in betas.items():
        filtered_candidates.append(((alpha, beta), prob))
filtered_candidates = sorted(filtered_candidates, key=lambda x: x[1],
 ↪reverse=True)
filtered_candidates[:30]
```

[13]:
```
[((1, 2176), 0.00040375441312789917),
 ((3, 32768), 0.00039661675691604614),
 ((1, 2056), 0.0003714263439178467),
 ((11, 32768), 0.0003408752381801605),
 ((9, 34824), 0.0003184061497449875),
 ((32, 4352), 0.00031538738403469324),
 ((16, 272), 0.00031525129452347755),
 ((2, 2176), 0.00031377002596855164),
 ((14, 32768), 0.0003067190991714597),
 ((48, 4096), 0.00030543468892574310),
 ((3, 34824), 0.00030264444649219513),
 ((12, 34816), 0.00030050426721572876),
 ((13, 32768), 0.00029943918343633413),
 ((16, 257), 0.00029497360810637474),
 ((14, 8), 0.0002943561412394047),
 ((3, 34816), 0.00029219873249530790),
 ((13, 8), 0.0002897172234952450),
 ((9, 34952), 0.00028816051781177520),
 ((2, 34816), 0.00028765015304088590),
 ((15, 2176), 0.00028703059069812300),
 ((2, 2056), 0.00028532557189464570),
 ((224, 1), 0.00028273626230657100),
 ((1, 34816), 0.00028091669082641600),
 ((3, 34952), 0.00028022192418575287),
```

```
 ((144, 4353), 0.0002791935548884794),
 ((32, 17), 0.00027744739782065153),
 ((48, 4352), 0.00027464982122182846),
 ((240, 272), 0.00027415436125011183),
 ((176, 4096), 0.0002709709224291146),
 ((15, 32904), 0.0002706355880945921)]
```

1.10 Перевірка ключів

[7]:
```
%%time

# 1/0.00015 = 6666 ~ 7000
# 7000 = 8 * 875
SAMPLES_NUM = 7000
# checking keys from 0..=((1<<16)-1)
keys = [i for i in range(0, MAX_16BIT_NUM + 1)]


def get_unique_alpha_beta_values(candidates):
    return set([alpha for (alpha, _), _ in candidates]).
  ↪union(set([beta for (_, beta), _ in candidates]))


# precompute values for computing u(k) value
precomputed_custom_mul = dict(
    [(x, [custom_mul(x, k) for k in keys]) for x in
  ↪get_unique_alpha_beta_values(filtered_candidates)])


# text -- 16bit numbers
def write_file(file, text: list[int]):
    with open(file, 'wb') as f:
        for x in text:
            f.write(x.to_bytes(2, byteorder='little'))
        f.close()


def read_file(file):
    text = []
    with open(file, 'rb') as f:
        data = f.read()
        for pair in zip(data[::2], data[1::2]):
            el1, el2 = pair
            text.append((el2 << 8) | el1)
        f.close()
    return text
```

```
def get_plaintext_filename(variant: int, index: int):
    return './key_check/pt_{0}_{1}.bin'.format(variant, index)


def get_ciphertext_filename(variant: int, index: int):
    return './key_check/ct_{0}_{1}.bin'.format(variant, index)


def generate_ciphertext(samples_num, index):
    random.seed(0)
    plaintext = [randrange(0, MAX_16BIT_NUM + 1) for _ in range(0,
  ↪samples_num)]

    filename_pt = get_plaintext_filename(VARIANT, index)
    filename_ct = get_ciphertext_filename(VARIANT, index)
    write_file(filename_pt, plaintext)

    subprocess.Popen('../lab1/data/heys.bin e {0} {1} {2}'.
  ↪format(VARIANT, filename_pt, filename_ct),
                     shell=True,
                     stdout=subprocess.DEVNULL, stderr=None).
  ↪communicate(timeout=10)
    ciphertext = read_file(filename_ct)

    with Pool() as pool:
        heys_1_round_enc = list(
            pool.map(lambda k: [heys_obj.mix_words(heys_obj.
  ↪substitute(x ^ k, ENC)) for x in plaintext], keys))

    return heys_1_round_enc, ciphertext


# \overset{u_k}{^} = |(X_1, Y): \{ \alpha \cdot X_1 \oplus \beta \cdot
  ↪Y = 0 \}| - |( X_1, Y): \{ \alpha \cdot X_1 \oplus \beta \cdot Y = 1
  ↪\}| = \text{# символів y} (\alpha \cdot X_1 \oplus \beta \cdot Y) -
  ↪2 \cdot |(X_1, Y): \{ \alpha \cdot X_1 \oplus \beta \cdot Y = 1 \}|
def check_candidate(heys_1_round_enc_texts, ct, alpha, beta):
    u_k = 0
    for (ct1, ct2) in zip(heys_1_round_enc_texts, ct):
        u_k += precomputed_custom_mul[alpha][ct1] ^
  ↪precomputed_custom_mul[beta][ct2]
    return abs(SAMPLES_NUM - 2 * u_k)
```

```
CPU times: user 223 ms, sys: 9.88 ms, total: 233 ms
Wall time: 232 ms
```

[8]:
```
%%time
CANDIDATES_THRESHOLD = 200

heys_1_round_enc, ciphertext = generate_ciphertext(SAMPLES_NUM, 1)
possible_key_candidates = {}
for (alpha, beta), _ in filtered_candidates:
    with Pool() as pool:
        success_rates = list(pool.map(lambda k:
  ↪check_candidate(heys_1_round_enc[k], ciphertext, alpha, beta),
  ↪keys))

        candidates_taken = CANDIDATES_THRESHOLD
        possible_max_candidates = []
        for max_u_k_value in range(max(success_rates), 0, -1):
            keys_with_max_u_k = [k for k, u_k in
  ↪enumerate(success_rates) if u_k == max_u_k_value]

            if len(keys_with_max_u_k) <= candidates_taken:
                possible_max_candidates += keys_with_max_u_k
                candidates_taken -= len(keys_with_max_u_k)
            else:
                possible_max_candidates += random.
  ↪sample(keys_with_max_u_k, candidates_taken)
                break

        possible_key_candidates[(alpha, beta)] =
  ↪possible_max_candidates
```

```
CPU times: user 1min 56s, sys: 13min 48s, total: 15min 45s
Wall time: 38min 45s
```

1.11 Результат атаки

Отримали наступний список із потенційними ключами серед яких є правильний ключ.

[15]:
```
sorted_key_candidates = {}

for (_, _), keys in possible_key_candidates.items():
    for key in keys:
        if key in sorted_key_candidates:
            sorted_key_candidates[key] += 1
        else:
            sorted_key_candidates[key] = 1

sorted_key_candidates = sorted(sorted_key_candidates.items(),
  ↪key=lambda x: x[1], reverse=True)
[(hex(key), count) for key, count in sorted_key_candidates[:100]]
```

```
[15]: [('0x2f43', 12),
       ('0x2d43', 11),
       ('0x2d47', 8),
       ('0x5f43', 8),
       ('0xaf43', 8),
       ('0x2543', 8),
       ('0x2943', 8),
       ('0x2e43', 8),
       ('0x5543', 8),
       ('0x2143', 8),
       ('0x6d43', 8),
       ('0xff43', 8),
       ('0xc043', 8),
       ('0x2743', 8),
       ('0xf43', 7),
       ('0x2b43', 7),
       ('0xbf43', 7),
       ('0x6f43', 7),
       ('0x5a43', 7),
       ('0x2c43', 7),
       ('0x4343', 7),
       ('0x6143', 7),
       ('0x7f43', 7),
       ('0xf243', 7),
       ('0xc143', 7),
       ('0x2df3', 7),
       ('0x2d48', 7),
       ('0x2d44', 7),
       ('0x2d4b', 7),
       ('0x2f4f', 7),
       ('0x2d4e', 7),
       ('0x340d', 7),
       ('0x2947', 6),
       ('0x2f47', 6),
       ('0x2d4d', 6),
       ('0x34cc', 6),
       ('0x9f43', 6),
       ('0x1a43', 6),
       ('0xba43', 6),
       ('0x5643', 6),
       ('0x643', 6),
       ('0xef43', 6),
       ('0xce43', 6),
       ('0x2643', 6),
       ('0x743', 6),
       ('0x1243', 6),
       ('0xc243', 6),
```

```
('0xc543', 6),
('0x5b43', 6),
('0x5e43', 6),
('0xb743', 6),
('0xbd43', 6),
('0x943', 6),
('0xc943', 6),
('0x143', 6),
('0x6043', 6),
('0x2843', 6),
('0xf943', 6),
('0x7d43', 6),
('0x9743', 6),
('0x1f43', 6),
('0xc343', 6),
('0x6743', 6),
('0xa91a', 6),
('0xfbd', 6),
('0xbfda', 6),
('0x2443', 6),
('0x5c43', 6),
('0xcf43', 6),
('0x2a43', 6),
('0x2d63', 6),
('0x8b8f', 6),
('0x2dd3', 6),
('0xcad0', 6),
('0x34eb', 6),
('0x2343', 6),
('0x2042', 6),
('0x2844', 6),
('0x2f45', 6),
('0x2d49', 6),
('0x2d4c', 6),
('0x2df7', 6),
('0x126a', 6),
('0x4fc3', 5),
('0x3be', 5),
('0x2047', 5),
('0x2247', 5),
('0x2447', 5),
('0x2547', 5),
('0x2a47', 5),
('0x2b47', 5),
('0x2e47', 5),
('0x5f83', 5),
('0x698d', 5),
```

```
('0x2345', 5),
('0xeecf', 5),
('0x342c', 5),
('0x1564', 5),
('0x4364', 5),
('0x2b4d', 5)]
```

1.12 Висновки

За допомогою реалізації другого практикуму із блокових шифрів по знаходженню останнього раундового ключа для шифру Хейса дізналися на практиці, як проводяться атаки даного штибу для більших криптосистем.